Engenharia Informática

Unidade Curricular de **Programação Distribuída**

Ano Curricular de 2020/2021

Trabalho Prático

Meta 2

Realizado por:

Diogo Lima	Nº 2014010835
Gabriel Gomes	Nº 2017018332
Sérgio Soares	Nº 2016014425

Índice

Introdução

No âmbito da unidade curricular de Programação Distribuída, foi proposto a realização de um projecto cliente servidor a simular uma aplicação de chat aplicando conceitos de Java RMI e API Rest.

Este relatório corresponde à meta dois e é uma expansão do projeto que foi desenvolvido na meta um.

Este relatório está dividido em seis partes:

A primeira é uma breve contextualização da meta dois. Depois são explicados em sequência a implementação do Java RMI, a aplicação autónoma e a API REST. De seguida são explicadas as decisões tomadas durante o desenvolvimento do programa e por último as funcionalidades que ficaram por implementar no projeto.

Tudo termina com uma breve conclusão em que se reflete no processo de desenvolvimento.

É importante realçar desde cedo e vai ser feito novamente no decorrer do relatório que o Java RMI foi usado para mais do que aquilo que era estritamente pedido no relatório. Como o programa desenvolvido na primeira meta não era o mais estável, foi tomada a decisão por parte dos programadores de modo a assegurar a integridade do programa.

1 - Conceptualização

Durante o desenvolvimento desta segunda meta do trabalho prático, as funcionalidades requeridas, ditaram a introdução de novas classes e interfaces. Estas vieram na maior parte substituir outras classes devido à simplificação que o RMI permite ao programador.

Foi prioridade corrigir alguns erros da primeira meta. Um desses erros era a incapacidade de correr mais de dois servidores em simultâneo devido à forma como se estava a proceder a atualização inicial assim que um servidor se ligava. Este erro foi corrigido e é possível ter quantas instância do servidor se quiser.

Foram criados para cada instância de um servidor um correspondente servidor RMI.

Para o servidor RMI foi criada uma classe e a sua interface remota (*RemoteServer*) e para os callbacks outra (*BroadcastReceiver*).

Como na primeira meta do trabalho não foi desenvolvida a base de dados, para a segunda meta o desenvolvimento seguiu essa abordagem sendo tudo localmente armazenado em cada servidor e atualizado por multicast.

Muitas das classes utilizadas para as threads foram eliminadas devido ao Java RMI e razões a serem explicadas mais abaixo e na secção da documentação das decisões.

Para a API REST foi usado um modelo fornecido nas aulas. Desta forma a WebSecurity configurado.

2 - Java RMI

Para esta segunda meta foi proposta a implementação de três funcionalidades tendo por base o conceito de servidores remotos RMI. Estas são as seguintes:

- Registar novos utilizadores.
- Enviar uma mensagem para todos os utilizadores que estão ligados ao mesmo servidor.
- Registar e remover observadores que irão receber notificações assíncronas.

Rapidamente se percebe durante a implementação do RMI que é uma forma extremamente simples e intuitiva de implementar comunicação entre servidor e cliente.

Para além de ser simples, é completamente transparente a parte de comunicação entre sockets. Esta comunicação está encapsulada através da interface *Remote*. Deste modo o utilizador pode aceder a métodos implementados na interface remota, que foi criada pelo servidor com o qual pretende comunicar e simplesmente chamar métodos que irão desencadear métodos no servidor destino.

Na implementação em questão, por razões de estabilidade dos sistemas apresentados na meta um, foi implementado Java RMI para todas as operações que requerem a normal comunicação via sockets TCP na meta um.

As interfaces implementadas para a realização do RMI e Callbacks foram:

- ***BroadcastListenerInterface*** - Interface responsável pelos métodos utilizados para os callbacks dos observadores registados.
- ***RemoteServer*** - Interface responsável pelos métodos dos servidor remoto que o *User* irá usar para comunicar com o *Server*.

Para cada uma das interfaces foi implementado um classe correspondente, que implementa os métodos declarados na interface:

- ***BroadcastListener*** (implements *BroadcastListenerInterface*): responsável pela implementação dos métodos de callback. Este é criado na classe *user*, que é passado por argumento no construtor para este ter acesso depois a métodos necessários para manter a interface de texto.
- ***ServerRMI*** (implements *RemoteServer*): responsável pela implementação dos métodos para comunicação RMI. Criado pela classe *server*, que é passada por argumento para que possam ser chamados métodos implementados no servidor.

3 - Aplicação Autónoma "Outsider"

Para o desenvolvimento da aplicação autónoma, que foi intitulada de "outsider". Esta corre independente de todos os outros projetos.

Em termos de comunicação uma thread que irá processar a receção de informação via multicast. Assim como os servidores comunicam entre si para se atualizarem e fazem várias vezes request de informação para quando um cliente se liga, esta aplicação irá ser capaz de fazer e de ouvir esses requests.

Quando estes request são feitos, a informação de todos os servidores é enviada numa string formatada que depois vai ser interpretada pelos recetores. Esta string contém os IPs, portos UDP e portos RMI, estes últimos que serão usados para conectar a aplicação ao servidor RMI em questão.

Esta aplicação tem de ser capaz de desempenhar as mesmas funções permitidas no servidor remoto, tais como:

- Registar novos utilizadores.
- Enviar uma mensagem para todos os utilizadores que estão ligados ao mesmo servidor.
- Registar e remover observadores que irão receber notificações assíncronas.

As classes criadas para esta aplicação são as seguintes:

- ***Outsider*** - Responsável pelo programa em si contendo o seu ciclo de execução. Engloba a interface de texto e assim como a interação com o utilizador.
- ***UpdateRMIThread*** - Responsável pela thread de atualização do *Outsider*. Tem um socket multicast que recebe a informação que os servidores mandam por multicast e informa a aplicação principal das opções e dos novos servidores remotos existentes. É capaz de ouvir e enviar pedidos de ocupação.

4 - API REST

Para esta segunda meta foi também proposta a implementação de três funcionalidades tendo por base o conceito de API REST/aplicação Spring Boot. Estas são as seguintes:

- Autenticar utilizadores.
- Obter as últimas *n* mensagens trocadas no servidor, sendo cada uma descrita pelo remetente, destinatário (utilizador ou canal) e pela mensagem em si;
- Enviar uma mensagem para todos os utilizadores que estão ligados ao mesmo servidor.

A API fornece a possibilidade de receber um pedido em qualquer tipo de linguagem e poder processá-lo sem problemas, de forma simples e de fácil compreensão.

A API está ligada ao servidor através do servidor RMI, fazendo todas as ligações através do mesmo. Para segurança da nossa aplicação, apenas a função de autenticação é permitida sem um *token*, todas as outras funcionalidades precisam de autorização para poderem ser acedidas, tal como pedido.

Para a receção e envio dos dados foram criados dois controladores que são as classes que interagem com o utilizador. Os dois controladores são:

- ***UserController*:** todos os métodos dentro deste controlador precisam de */user* antes do mapping do método, pedido através do *@RequestMapping*. Este controlador tem dois métodos:
    - **public User login(@RequestBody UserAuthentication user):** o primeiro, e também o único que pode ser executado sem o *token* de segurança. Este é um método *post* e é responsável por fazer o login do user, enviando os dados recebidos para o servidor RMI, depois de enviar os dados para o RMI recebe a confirmação da validade e gera um *token* para o utilizador poder aceder ao resto das funcionalidades;
    - **public String broadcast(@RequestBody (...), @RequestParam (...)):** este é um método *post* e é responsável por enviar uma broadcast message para todos os utilizadores online no mesmo servidor, recebe a mensagem a enviar do body, e o username que quer enviar a mensagem dos params;
    - **public String logout(@RequestParam (...)):** este é um método *post* e é responsável por executar um logout seguro ao utilizador, do qual recebe o *username* no param do URL.
- ***MessagesController*:** este controlador contém um método que é responsável por apresentar as últimas *n* mensagens do servidor em que o utilizador se encontra. O método é o seguinte:
    - **public void lastMessages(@RequestParam (...)):** este é um método *get* e é responsável por mostrar as últimas mensagens do servidor. Recebe o número de mensagens desejados no param, ou então vai buscar as últimas 50, valor por *default*.

5 - Documentação das Decisões

Durante a realização do trabalho, foi necessário a tomada de certas decisões, de modo poder prosseguir com guidelines definidas para como seria o resto do desenvolvimento.

Esta é a decisão mais importante tomada nesta meta. Para garantir a consistência do código e garantir que tudo corria como deve, foi tomada a decisão de mudar toda a essência da comunicação entre *User* e *Server* para que esta possa correr como RMI. Deste modo é garantido que pequenos problemas que surgiram na primeira meta com as ligações TCP não irão aparecer. Como foi dada a opção de criar um projeto de raiz para a segunda meta, sem serem necessárias todas as funcionalidades anteriores, é suposto isto não ser um problema.

Foi também corrigido na primeira um problema que existia no programa da primeira meta que impedia que mais de duas instâncias do servidor estivessem em simultâneo a correr. Este problema foi neutralizado.

Um User assim que se cria, regista-se como observer do servidor ao qual se ligou. Esta decisão foi tomada no sentido em que fazia mais sentido e simplificava o processo de callback imenso. Em vez de cada utilizador se registar num chat privado ou num canal, o utilizador observa o servidor que depois tem a função de notificá-lo quando for necessário. Isto significa que a única forma de remover o observador de algo é desligando saindo de um canal, visto que os chats privados não podem ser apagados, ou então desconectar o utilizador, que automaticamente remove o observador.

Como o programa corre como aplicação consola e funciona com interface de texto, a integridade do elemento visual quando as notificações assíncronas vindas dos callbacks são chamadas e se imprime algo no ecrã, onde o User ficava interrompido pela nova notificação impressa no ecrã.

Para a aplicação Outsider, como esta tem de ser capaz de criar utilizadores, mas estes são uma aplicação independente e têm funcionalidades independentes, o utilizador que vai ser criado pelo Outsider estará automaticamente offline.

O Outsider pode ser criado com ou sem servidores a correr. Quando este inicia, faz um pedido de ocupação por multicast, que caso algum servidor esteja ativo irá ouvir e responder com toda a informação incluindo IP, e mais importante o porto RMI que se vai usar para ligar ao servidor remoto. Deste modo, não é necessário passar nenhuma lista de endereços IP à aplicação quando esta inicia, sendo completamente autónoma.

Ao contrário dos utilizadores, a aplicação Outsider consegue ouvir vários servidores ao mesmo tempo, mas para tal tem de se registar neles. Apenas aí passará a ouvir as notificações através dos callbacks. É também possível manualmente remover observers de servidores.

6 - Funcionalidades Por Implementar

Não ficou nada por implementar na segunda meta.

Conclusão

Com os conceitos estudos durante as aulas da unidade curricular de Programação Distribuída e a realização desta segunda meta do trabalho prático, foi concluído que no desenvolvimento de um programa como estes a estruturação e organização da troca de mensagens entre os dois processos é essencial. Várias vezes durante o desenvolvimento foram sofridos atrasos derivados a estas pequenas inconveniências.

O conceito de Java RMI é extremamente intuitivo e versátil, permitindo a comunicação via TCP de forma transparente ao utilizador, que pode simplesmente chamar métodos implementados numa interface remota e comunicar com o servidor. Esta forma é tão intuitiva que até foram feitas mais funcionalidades com ela tal o conforto e segurança que é fornecido.

Assim como o RMI o API REST segue o mesmo princípio de simplicidade, permitindo apenas a declaração de uma interface e os seus métodos que se deseja implementar.

Por último, conclui-se que, tal como em outras linguagens orientadas a objetos, a existência dos mesmos durante a programação é algo que torna a programação extremamente mais intuitiva.